MySQL Deserves a Double Take

Reuven M. Lerner

Abstract

What you don't know about MySQL could hurt you.

In early 1995, when Windows 95 was still vaporware and Red Hat was the upstart, user-friendly alternative to Slackware, I worked for the “Pathfinder” site at media giant Time Warner. Like all media companies, Time Warner realized that the Web was going to take off, but wasn't sure just how that would happen.

So, it hired a bunch of programmers and designers, and gave us the opportunity to experiment with different types of designs and applications. It was a wonderful job, with some creative, smart and interesting people. And, during my time there, I created all sorts of applications—quizzes, mail auto-responders, games, search engines and even a personalized version of *Money* magazine's “Best Cities” rankings.

As the applications I built became increasingly sophisticated, it became obvious that the text files I often used for data storage and retrieval were neither efficient nor flexible enough for a site as large and popular as ours. Finally, someone introduced me to our newly hired database guru, who taught me about the wonders of relational databases and SQL. I was hooked, and I enjoyed working with the database server that we had installed.

When I wanted to reap the benefits of SQL on my Linux box at home, my options were, unfortunately, limited. I found a number of abandoned open-source database projects, but nothing that was as powerful as Time Warner's Sybase server or even in the same league.

So, you can imagine my delight when I discovered MySQL. No, it didn't do all the things that Sybase did, and it wasn't released under an open-source license. But, it was free of charge, it was easy to install and it had enough features to keep people like me relatively happy. Internet service providers felt similarly, and began to install it on their systems—first as a competitive advantage over their rivals, and then because everyone else was including it in the base configuration.

Fast-forward more than a decade, and MySQL is by far the best-known open-source relational database. Monty Widenius and David Axmark, whom I met back when they were the only full-time MySQL programmers, are now at the top of a large corporate pyramid. MySQL AB now distributes its products under the GNU General Public License (GPL), with a closed-source license available to those who require it. It runs, as always, on a very large number of different operating environments. And, it is still developed at a feverish pace by people around the world, who submit patches and suggestions.

This is the first of three articles on open-source databases. This month, I discuss MySQL, including its use, features and problems. The next article will include a similar analysis of PostgreSQL, and the third article in the series will compare the two databases.
Starting with MySQL

One of MySQL's claims to fame is the ease with which people can get started using it. And, indeed, when you compare MySQL with many commercial databases, it is strikingly simple. You install it (typically with an RPM or Deb, but compiling it from source is also straightforward), and start up the database server with `safe_mysqld`. (You also could use the plain `mysqld` command, but then you wouldn't benefit from some of the behind-the-scenes housekeeping that `safe_mysqld` offers.)

Once you have started the server, you can create one or more databases. (I admit it is somewhat confusing that MySQL is often referred to as a database when, in fact, it is a database server, offering you the chance to create one or more databases. Each database contains one or more two-dimensional tables.) To create a database, use the `mysqladmin` program:

```
mysqladmin create testdb
```

It is quite possible, depending on your configuration, that the above command worked without a hitch—particularly if you are logged in as the root user under Linux. However, your system administrator might have (wisely) decided to set a password for the MySQL root user, in which case, you need to type:

```
mysqladmin -p create testdb
```

The `-p` option tells `mysqladmin` that you want to enter a password for this account. You also can specify the root user, or any other user, with the `-u` option, as in:

```
mysqladmin -u mysqlroot -p create testdb
```

Once you have created a database, you then can connect to it with the `mysql` client program:

```
mysql -u mysqlroot -p testdb
```

Notice that I'm once again specifying a user name and that I want to enter a password. I return to the subject of permissions below; for now, we assume that this combination works.

In the client, you can issue any SQL command you want, and it will be executed immediately. For example, we can create a new table:

```
CREATE TABLE Classes (
    class_name    TEXT    NOT NULL,
    room_number INTEGER  NOT NULL,
    starting_date DATE NOT NULL,
    ending_date   DATE  NOT NULL,
    instructor       TEXT    NOT NULL
);
```

One of the problems with the above table is that it lacks a unique primary key. This makes it difficult to refer to the Classes table from another table. We could use the name assigned to the class by the university's registration system, but there is no guarantee that this will be unique. Moreover, what will we do next year, when a class of the same name is offered? For this reason (among others), it's the norm to create an "artificial" primary key, one whose purpose is to identify a row within the database uniquely.
In MySQL, we can do this most easily with the AUTO_INCREMENT keyword. For example:

```sql
CREATE TABLE Classes (
    class_id         INTEGER AUTO_INCREMENT,
    class_name    TEXT    NOT NULL,
    room_number INTEGER  NOT NULL,
    starting_date  DATE NOT NULL,
    ending_date   DATE  NOT NULL,
    instructor       TEXT    NOT NULL,

    PRIMARY KEY(class_id)
);
```

If we want, we can INSERT a row into Classes with an explicit integer value for class_id. The fact that class_id is defined as a primary key means that it is both indexed and guaranteed to be unique. But, if we fail to enter an explicit value for class_id, MySQL inserts a new value into the column, giving us a primary key value for the new class without having to calculate it ourselves.

The above table definition shows a few of the many data types MySQL offers. MySQL offers many traditional data types, such as NUMERIC and VARCHAR, but it also includes a number of signed and unsigned numeric types (for example, TINYINT, SMALLINT, MEDIUMINT, INT and SIGINT), a number of CLOB/BLOB types (such as, CHAR, BINARY, BLOB and TEXT), and several having to do with dates and times (DATE, DATETIME and TIMESTAMP). There are also ENUM and SET types, allowing you to work with nonstandard sets of enumerated data.

MySQL also offers a wide variety of operators, from simple string-concatenation, to date extraction, to one of my favorites, the CASE statement, which lets you place if-then logic inside of a query.

In addition, MySQL offers a system for full-text search. This means you can store text inside of TEXT columns in your tables, and then identify the column (and retrieve the text) without having to index it yourself.

If the included suite of functions doesn't suit your needs, you can always write one of your own. Recent versions of MySQL also offer the ability to create a stored procedure or function, which provides both increased speed and centralized control over commonly used functions. Stored procedures also can be invoked automatically when particular events occur, known as a trigger in database parlance. You also can write new functions in C or C++, loading them into MySQL at runtime.

**Table Types**

So far, MySQL sounds like a nice, flexible relational database. You might be surprised, however, to find that there is a huge amount of pent-up frustration, and even hostility, toward MySQL in the Open Source and Database communities. Just look for a recent story on Slashdot about MySQL, and you will see many comments indicating that PostgreSQL, Firebird or nearly any other option would be a better solution.

Part of this stems from a time-honored tradition of rivalry in the computer world, and particularly in the Open Source community. Over the years, we have seen fights between Emacs and vi, Perl and Python, Linux and BSD, and countless other pairings.

*But, part of the animosity toward MySQL stems from several design decisions that the authors made*
early on. For example, documentation for an old version of MySQL said that foreign keys are really unnecessary, and that such integrity checks could (and should) be handled in the application, rather than in the database. Many experienced database people see this and don't know whether to laugh or cry. The primary reason for using a database is for its reliability, not speed, and adding foreign-key checks is an easy way to increase the reliability of inserted data.

Similarly, old versions of MySQL failed to lock tables. If you wanted to be sure that no one would write to a table from which you were reading (or to which you were writing), you needed to lock the table explicitly at the application layer. Given the many years of research that had gone into row-level locking (and even more-advanced systems, such as mutlilversion concurrency control), this seemed to many like a step backward.

MySQL's solution to these problems has been a novel one. Rather than add these features to the existing (MyISAM) table structure, it made it possible to choose from a number of different table structures, each with its own set of trade-offs. Much as Linux system administrators can choose from a variety of filesystems, MySQL administrators and programmers can choose from several different storage engines.

This approach has some problems, of course. The biggest problem from my perspective is that MyISAM remains the default storage engine, which means that many users effectively choose to go without foreign keys and sophisticated locking due to ignorance. Many other storage engines seem to be of more limited use or for particular applications, such as MEMORY (for in-memory databases), BDB (Berkeley DB-based) tables and even FEDERATED (for tables on remote servers).

A very popular storage engine, InnoDB, has a different problem associated with it—the company that develops InnoDB was purchased by Oracle earlier this year. This may have no effect on MySQL's open-source distribution, because Oracle continues to make InnoDB available under the GPL. But, it has raised some questions regarding MySQL's commercial version, given that an essential part of the commercial-grade toolbox is now owned by a major database rival.

Much has been made about MySQL's fast performance over the years, with little or no tuning of the server. The truth is a bit hazier than that; although MySQL is undoubtedly a fast database, many of those tests were made using MyISAM tables, which are inherently faster because of their lack of locks and integrity checks. (As an analogy, I often say that it's faster to leave your house without locking the door, but the extra speed is usually not worth the risk.)

**Scalability**

Many of the features in recent versions of MySQL have been aimed at corporate customers, whose license purchases are helping drive MySQL development forward. One of the biggest bottlenecks that a database administrator can face, particularly as the data grows in size, is disk speed. Recent versions of MySQL thus provide both tablespaces (that is, allocation of disk space on a per-table basis) and partitions (that is, division of a table across several filesystems). Tablespaces are available only with InnoDB tables, but partitions are available for all storage engines. Moreover, tables can be partitioned based on column values, using a hash function to decide into which partition a particular row should be placed.

Another important aspect of MySQL has been replication and backup. These are crucial features for enterprise clients, who need their data to be available all the time and to have backups available at a
moment's notice. The latest versions of MySQL have improved the replication engine and have also made it more flexible, making it possible to replicate tables even on a per-row basis.

Another feature I have been waiting to see for some time is Unicode support. Although not all string and regexp operations work with Unicode, this is a big boon to those who work with multiple languages.

Community

Perhaps the biggest asset that MySQL has going for it is a very large, very active community of users and developers. The sheer number of books, Web sites, mailing lists, help forums and code snippets for MySQL is overwhelming.

For its part, MySQL AB has been doing an admirable job of updating the documentation on a regular basis and of moving forward with new features at an impressive rate. (This demonstrates that although open-source software can often be written by volunteers, having paid professionals work on a project can speed it up immensely.) In particular, I am impressed by the on-line documentation, which includes not only numerous examples, but also intelligently placed links to related subjects.

Conclusion

MySQL has grown up quite a bit since I first began to use it more than ten years ago. Some of its quirks, such as using MyISAM tables by default, continue to rankle serious database users who would like to see transactions and foreign keys everywhere. But, especially with versions 5.0 and 5.1, MySQL is looking like a database that can advertise its depth of serious features, rather than claim its main advantage is speed.
How we use DOUBLE, DECIMAL, FLOAT and differences between each other. Negative numbers are also included. In this article, we discuss how to choose suitable mixed decimal data types and options to store negative numbers in MySQL. We specifically emphasize on decimals, double and floats and differences between them. Double. Double is a data type that is used to store floating-point numbers. Therefore double can represent negative or positive mixed decimal numbers (example 32.4334) to a limited accuracy.

Syntax. FLOAT(N). Table 1. Double. Where N is any number from 24 to 53. Double is an approximate numeric data value since some numbers given in decimal are approximated when converted to the double data type. I have a PHP form that takes user input and stores it in a MySQL database. I have the variables designed like this: $description1 = mysql_real_escape_string($_POST[description1]); The SQL INSERT statement is pretty standard: INSERT INTO orders (description1) VALUES ('$description1'); This works fine for data containing single quotes, but anything with double quotes gets cut off where the double quotes start. Only the data up to the double quote gets written to the database. I’m sure I’m missing something obvious, but where? Thanks for your help!

References

Download ppt “MySQL Vs. PostgreSQL comparison from a laymanâ€™s perspective By: Bhalchandra.” Similar presentations.